CHAPTER
SIX
SHARED VARIABLES

Most of the models and languages we discuss in this book use explicit processes for the primitive, concurrent-processing object. These systems package the communications of these processes into some form of message and arrange the delivery of these messages to the right destinations. But what lies below messages? Nancy Lynch and Michael Fischer argue that sharing is the foundation of interprocess communication. They base their Shared Variables model on communication through reading and writing of shared variables. They claim three major advantages for their approach: (1) Variable sharing is a close reflection of computer hardware; (2) The frequency of reading and writing shared variables is an excellent metric of the complexity of distributed algorithms; and (3) Shared variables are primitive — all other "realizable" distributed models can be described in terms of shared variables. Lynch and Fischer's primary concerns are the complexity and correctness of distributed algorithms. Their system can be used to model not only the information-transfer aspects of distribution but also the protocols of communication.

Processes and Shared Variables

The Shared Variables model has two kinds of objects, processes and shared variables. Processes compute independently and asynchronously. They communicate only by reading and writing shared variables. A process that reads a shared variable obtains its value; a process that writes a shared variable changes its value. A variable may be shared by two or more processes. Figure 6-1 shows the variable sharing of several processes. In the figure, processes P and Q share variable

W, R and S share variable x, P, Q, and R share variable y, and all four processes share variable z.

A model based on shared storage would seem to be inappropriate for coordinated computing. After all, the criterion that distinguishes distribution from mere concurrency is the absence of shared state. Nevertheless, this is a distributed model. Lynch and Fischer argue its relevance by stating [Lynch 81, p. 19]:

> At the most primitive level, something must be shared between two processors for them to be able to communicate at all. This is usually a wire in which, at the very least, one process can inject a voltage which the other process can sense. We can think of the wire as a binary shared variable whose value alternates from time to time between 0 and 1. . . . setting and sensing correspond to writing and reading the shared variable, respectively. Thus, shared variables are at the heart of every distributed system.

Shared Variables is an attempt to model distributed systems at the most primitive level. It explicitly rejects the high-level facilities that programming languages provide. Operations such as implicit synchronization, clock interrupts, and message and process queueing are important for building real systems. However, such facilities are not primitive—they can all be described by a more fundamental mechanism, the shared variable. Describing an algorithm without the aid of such high-level facilities forces the writer to make the algorithm's protocols explicit; it permits the inherent costs of particular communication patterns to be quantitatively analyzed. Furthermore, shared variables are a known technology—the construction of a system described solely in terms of shared variables is straightforward.

A typical shared variable in the Lynch-Fischer model stores only a single, small value. In using this model for analyzing distributed systems, one ought to be limited to only a few such variables. However, nothing in the system design precludes modeling a large shared memory using many shared variables.

Each process in the Shared Variables model is a countably-infinitely-branching, nondeterminate, Turing-equivalent automaton. We explain the meaning of this phrase later in this section. For the moment, we assume that each

Figure 6-1 Processes and shared variables.

process is running some high-level language on a machine with an infinite memory and an unbounded random-number generator.

Like a Turing machine, each process has an internal state. At each computational step, a Turing machine reads its input tape, writes a value on its output tape, and enters a new state. Changes in the storage tape reflect changes to the Turing machine's state. Similarly, at each computational step a Shared Variables process selects a shared variable to read, reads that variable, writes a new value for that variable, and enters a new state. This entire operation is an atomic action—no other process can access the variable during the update, and this process cannot access any other variable in determining the update value. The process of checking the value in a variable and then updating it is called *test-and-set.*

Test-and-set allows the computation of an arbitrary function between the reading of the test and the writing of the set. For example, a process can read a numeric shared variable, compute its largest factor, and write that value back into the variable. Computing this factor may be a complex computation. Nevertheless, no other process can access that shared variable during the update.*

Despite the similarity of names, the Lynch-Fischer test-and-set is a much more powerful instruction than the standard test-and-set instructions found on many computers. In conventional computers, reading and writing of storage are performed on separate cycles. Other processes or interrupts can interleave with these actions. Some computers combine a limited form of reading and writing in instructions that perform actions such as "increment this location and skip the next instruction if the result is zero." Allowing an arbitrary computation between reading the shared variable and writing the new value may seem too unrealistic. The Lynch-Fischer test-and-set requires a semaphore or lock on each variable to keep other processes from accessing it during the computation of the update.

The full test-and-set instruction is too powerful to faithfully model conventional computers. For such systems we use a subset of the full class of shared variable processes, the read-write processes. If a read-write process reads a value, it must write back that same value. If such a process writes a value, both its new state and the value written must be independent of the value read. That is, in a single, indivisible step, a read-write process can appear to either read or write a variable, not both. Hence, every read-write process is a test-and-set process, but not every test-and-set process is a read-write process.

In the Lynch-Fischer model, processes never fail. More specifically, a process that starts to update a variable finishes that update. Thus, every process has a valid response to every possible value it might read from a shared variable. Furthermore, every shared variable eventually unlocks. Though processes cannot stop in the middle of updating a shared variable, they can stop between updates. These restrictions parallel the limits that mutual exclusion problems place on

* However, as this model lacks a notion of time, this use of "during" is somewhat misleading.

stopping inside a critical region. Just as Dijkstra assumed in his development of Dekker's solution (Section 3-2) that processes do not stop in critical regions, Lynch and Fischer assume that processes do not stop while updating.

This model is time-independent. A process cannot refer to the amount of time it has been waiting for a variable to change. It can only intermittently check to see whether the variable really has changed—somewhat akin to busy waiting. Of course, lacking "time," processes do not compute at any particular speed. One is sure only that a process that has not halted will eventually execute its next step.

The model's exclusion of time-dependent behavior is both a limitation and an advantage. Time is important in real systems. For example, clocks and interrupts are fundamental to operating systems. Therefore, relationships such as mainframe/terminal communications (where the two processes depend on having similar clock speeds) and time-outs (where a computation can be aborted after a specified duration) cannot be described in the Shared Variables model. On the other hand, results based on this model are, in some sense, stronger results because they do not depend on temporal assumptions.

As we have said, each process in the Lynch-Fischer model is a countably-infinitely-branching, nondeterminate, Turing-equivalent automaton. A Turing-equivalent automaton is a general computing device (Section 1-1). It can compute any function that can be coded in a high-level language like Pascal or Lisp. A bounded nondeterminate automaton can take each of several paths at a given choice point. An infinitely-branching nondeterminate system can also take several paths. However, it can select an infinite number of different paths at a single choice point. A countably-infinitely-branching automaton is restricted to having choice points with only a countably infinite number of choices.* Each of these extensions allows us some behavior [the computation of some (multivalued) functions] that the earlier class did not have.

Unbounded nondeterminacy is a mathematically unorthodox assumption. Most mathematical models of computation that allow nondeterminacy place a finite bound on the branching factor. Lynch and Fischer chose unbounded branching for their model because they wanted a single process to be computationally equivalent to a set of processes.

Lynch and Fischer prove two fundemental theorems about concurrent computation (given the assumptions of the model). The first is that two asynchronous, determinate automata together can produce infinitely-branching, nondeterministic behavior. This is achieved by having the first send a message to the second, counting until it receives a reply. Since the model does not limit the relative speed of processes, we cannot bound the delay between sending the request and receiving the answer. However, since the second process will *eventually*

* A countably infinite set can be placed in a one-to-one correspondence with the integers. This is a small infinite set. Larger infinite sets include sets such as the real numbers and the set of all functions from the reals to the reals.

progress and answer our request, the algorithm always terminates. This varying delay is effectively an unbounded random number and is sufficient to make the system unbounded and nondeterminate.

The second theorem states that any finite set of unbounded nondeterminate processes is equivalent to some single process. This is true because a single process can simulate the set of processes by interleaving their steps. Given the unbounded delay between process steps, this theorem would be false if the automata were not infinitely-branching nondeterminate.

Lynch and Fischer also give a formal definition of what it means for a distributed algorithm to *solve a problem.* Their definition treats the actions of a set of processes as a permutation of the actions of the individual processes. A system solves a problem if every possible permutation yields a *solution state.*

Examples

The Shared Variables model is a perspective on distributed computing. Our examples illustrate the effect of this perspective. Models are not programming languages. Therefore, we cloak the semantics of the Lynch-Fischer model in the syntax of a programming language. We extend Pascal to describe shared-variable algorithms with the following conventions: A process that shares an external variable declares that variable as a **shared** variable. The declaration can specify an initial value for the variable. We create processes with a **process** declaration. We take liberties with the syntax of **const** definitions and with the evaluation of expressions in type declarations.

Semaphores As an example of the full test-and-set capability of the Shared Variables system we model a general semaphore. This semaphore controls a resource that can be simultaneously shared by up to ConcurrentUsers different processes. The semaphore is a shared variable semvar. Process Q executes the P or V operations on semvar. The brackets ≺≻ denote atomic actions. Each process that uses the semaphore has its own copy of the code for P and V.

```
process Q;
shared var semvar: integer (initial ConcurrentUsers);     -- imported variable
var resourcebusy: boolean (initial false);     -- Resourcebusy, a local variable (not
                                                  shared) of process Q, is true when
                                                  the resource is both sought and
                                                  unavailable.

procedure P;
    var local_semvar: integer;
    begin
        local_semvar := semvar;
```

```
          if local_semvar = 0 then            -- read the shared variable
               begin
                    resourcebusy := true;
                    semvar       := 0         -- write a new value
               end
          else
               begin
                    resourcebusy := false;
                    semvar       := local_semvar − 1     -- write a new value
               end
     end;

procedure V;
     begin
          semvar := semvar + 1
     end

-------------------- main program ------------------
begin
       ⋮
     repeat ≺P≻ until not resourcebusy;      -- request the resource
       ⋮
     ≺V≻                                     -- release the resource
       ⋮
end -- process Q
```

We use local procedures P and V to emphasize that the Lynch-Fischer model allows the reading and the writing of the shared variable to be separated by an arbitrary computation (except that these procedures cannot access any other shared variables). Q can inspect its own variable, resourcebusy, to determine if it has gained access to the resource controlled by the semaphore.

The full test-and-set primitive is unrealistically powerful. We have confined the remaining examples of the Lynch-Fischer model to read-write processes: processes that cannot both read and update a shared variable in a single atomic step.

Our next example demonstrates that read-write processes suffice for process synchronization. We describe a solution by Gary Peterson [Peterson 81] of the fair mutual-exclusion problem. His solution is simpler than Dekker's algorithm (Section 3-2). Peterson has the two processes, P and Q, communicate through three shared variables: turn, p_enter, and q_enter. Turn is a priority flag. In case of conflict, turn contains the process identifier of the next process to enter the critical region. The processes use p_enter and q_enter to show readiness to enter the critical region. Variable turn is read and written by both processes while each

Figure 6-2 Shared variables for mutual exclusion.

process sets the value of its enter variable and reads the value of its partner's enter variable. Figure 6-2 shows the information flow between the processes through the shared variables. P_enter is true when either P is in its critical region or wants to enter its critical section. A corresponding program specifies Q.

```
process P;
shared var
        turn              : process_id (initial p_id);    -- imported variables
        p_enter, q_enter  : boolean (initial false);
begin
    while true do
     begin
        p_enter := true;    -- declare intention to enter
        turn    := q_id;    -- ensure that Q can enter its critical region when
                               P's turn is over
        while q_enter and turn = p_id do
            skip;           -- wait until P's turn or Q is not in its critical
                               region
        p_enter := false;   -- withdraw intention to enter concurrent region
    end
end
```

Protocols One feature of Shared Variables is that it can model primitive protocols for interprocessor communication. A *protocol* is an organizational framework for communication: a mapping between the ordering of symbols and their interpretation. In computer systems, typical protocols specify that certain strings of bits are to have meanings such as "I want to send a message" and "Do you have a free buffer?" and replies to such requests. Protocols are particularly important when the communication channel is shared between several logical connections. In that case, the protocol identifies and orders the pieces of a message, keeping it from being scrambled with other messages.

Our next example presents a message transmission protocol that uses a single shared variable. Lynch and Fischer describe some of the difficulties of shared-variable communication [Lynch 81, p. 24]:

> The way in which processes communicate with other processes and with their environment is by means of their variables ... Unlike message-based communication mechanisms, there is no guarantee that anyone will ever read the value [in a variable], nor is there any primitive mechanism to inform the writer that the value has been read. (Thus, for meaningful communication to take place, both parties must adhere to previously-agreed-upon protocols. ...)

The key idea is that any pattern of communication can be achieved by reading and writing a shared variable according to appropriate protocols. To illustrate this idea we consider the problem of message communication between two read-write processes, P and Q. Assume that P and Q share variable pq_var that can store integers between $-n$ and n. P and Q wish to imitate a bidirectional message transmission system, where messages can be up to message_limit words long.

P and Q communicate by obeying the following protocol. P assigns only positive numbers to pq_var; Q only negative numbers. Both processes can set pq_var to 0. The numbers $n-1$ and n [respectively $-(n-1)$ and $-n$] are the "message initiation header" (pm_header) and the "end of message" (p_eom) for P (respectively, Q). We use the $n-2$ remaining values for encoding the message.

P stores the number $n-1$ (pm_header) in pq_var to show that it wishes to send a message. Q responds by writing a 0 (okay_to_send), indicating readiness to accept the communication. After seeing the value okay_to_send, P writes the body of the message in the variable, one word at a time. After each word, Q responds with a -1 (q_acknowledge). That is, P writes the first value in the message and Q replies with q_acknowledge. When P sees the q_acknowledge, it writes the second word and waits for Q's next acknowledgment. This continues until P has sent the entire message. At that point, P stores p_eom in pq_var. Q responds with okay_to_send. On seeing okay_to_send, either process can start another communication by placing its message initiation header into the variable. P and Q execute the corresponding protocol to send messages from Q to P. Variable pq_var is initially okay_to_send.

Problems arise when both processes try to write the message initiation header at about the same time. P and Q must recognize that writing the header does not guarantee the right to transmit. A process that writes the header and then reads the header of its partner does not send, but instead accepts the next message. When a process writes the header, it waits until its partner either accepts transmission (okay_to_send) or demands the channel for itself (the partner's message initiation header).

We present the program for P. The program for Q is analogous.

```
const
    qm_header      = −(n−1);    -- Q's message header
    pm_header      = n−1;       -- P's message header
    q_eom          = −n;        -- Q's end of message
    p_eom          = n;         -- P's end of message
    q_acknowledge = −1;         -- Q's response to message data
    p_acknowledge = 1;          -- P's response to message data
    okay_to_send   = 0;
type message = array [1 .. message_limit] of integer;
        -- Messages can be up to message_limit words long.

process P;
    shared var pq_var: integer (initial okay_to_send);      -- imported variable
        . . . <local declarations> . . .

    procedure receive;
    var
        in : message;     -- the message
        k  : integer;     -- the number of the words in the message
    begin
        if pq_var = qm_header then               -- Q has signaled that it wants
        begin                                       to send a message.
            pq_var  := okay_to_send;
            k       := 0;
            while pq_var ≠ q_eom do              -- repeat until Q signals it is
            begin                                   done
                while pq_var ≥ 0 do skip;        -- wait for response from Q
                if pq_var ≠ q_eom then
                begin
                    k       := k + 1;
                    in[k]   := pq_var            -- get the value
                    pq_var  := p_acknowledge;
                end
            end;
            pq_var := okay_to_send;              -- Acknowledge that the entire
                                                    message has been received.
            act(in,k);                           -- act on the message
        end
    end; -- receive

    procedure send (var out: message; k: integer);     -- send a message,
                                                          out[1..k]
    var
        i               : integer;
        ready_to_send : boolean;    -- Have we requested communication yet?
    begin
        ready_to_send := false;
        while not(ready_to_send) do
        begin
            while pq_var ≠ okay_to_send do receive;    -- Q wants to send.
            pq_var := pm_header;
            while pq_var = pm_header do skip;          -- waiting for
                                                          acknowledgment
            if pq_var = qm_header then receive         -- cannot send until
            else ready_to_send := true                    okay_to_send is in
                                                          pq_var
        end;

        for i := 1 to k do
        begin
            pq_var := out[i];                          -- write the next word
            while pq_var ≠ q_acknowledge do skip       -- wait for signal
        end;
        pq_var := p_eom;                               -- specify end of
                                                          message
            while (pq_var ≠ okay_to_send) and          -- wait for
                  (pq_var ≠ qm_header) do skip            acknowledgment
    end; -- send

begin
                  ⋮
  -- the code for P, including occasional checks to see if a message is waiting
     and calls to send messages as needed.
                  ⋮
end -- process P
```

This example emphasizes the fact that the only way to tell if a value has been read from a shared variable is to receive an acknowledgment of that reading.

We measure program efficiency by considering the quantity of a resource that it uses (proportional to its input) (Section 1.3). For example, the protocol program uses a shared memory cell that can accommodate $2n$ different values; the system writes more than $2k$ n-bit messages to transmit kn words of information. These bounds can be improved. For example, Burns [Burns 80a] has shown that a subtle extension of the work of Cremers and Hibbard [Cremers 79] allows the variable to be limited to as few as three values. Exercise 6-9 asks for an improved

Figure 6-3 A long distance communication.

protocol that uses fewer writes to send the same amount of information. The Lynch-Fischer model is particularly well suited for use in this kind of analysis.

Communication delay Shared variables provide failure-free communication between processes. Of course, many real world applications are difficult precisely because individual messages can become garbled or lost. One technique for modeling noisy communication between two shared-variable processes is to use a *channel process*. Imagine two distant processes, P and Q, that communicate through the shared variable x (Figure 6-3). Of course, with this arrangement P and Q have errorless communication. Anything one writes in the shared variable can be correctly read by the other. To model noisy communication, we introduce a *channel process* C that shares variable y with P and variable z with Q (Figure 6-4). C's task is to take the values written by P in y and to write them into z for Q, and to take the values written by Q in z and to write them into y for P. C is programmed to be noisy. That is, now and then it "makes a mistake"—writes the wrong value for a communication, drops a message, or inserts a spurious message. C's pattern of misbehavior can model the channel's intended noise pattern.

We assume that C has a function noise from channel values (integers) to channel values. Noise(x) is usually x, but sometimes it is something else. The code for this function reflects the particular pattern of noise being modeled. The program for the channel is as follows:

```
process channel;
shared var y , z: channel_value (initial 0);     -- imported variables
var yrecent, zrecent: channel_value;
begin
    yrecent := y;
    zrecent := z;
    while true do     -- using a guarded command
```

Figure 6-4 A channel process.

```
          y ≠ yrecent →
               z := noise(yrecent);   yrecent := y;   zrecent := z
     □
          z ≠ zrecent →
               y := noise(zrecent);   yrecent := y;   zrecent := z
     □
          y = yrecent and z = zrecent →
               skip
end -- channel
```

Election algorithms One use of the Lynch-Fischer model is to analyze the complexity of algorithms for distributed systems. In this next example we present a simple analysis of a distributed algorithm.

A common architecture for small distributed systems is a ring. In a ring, processes are arranged in a circle. Each process communicates directly with its left and right neighbors. Figure 6-5 shows a seven-element ring network. Each process in the ring shares a variable with each of its "next-door neighbors."

Sometimes a ring of processes must establish a chief (king) process. This process manages some aspect of the general system function. We assume that each process has a unique positive integer (its *rank*) and that the process with the largest rank ought to be the *king* (Figure 6-6). Process ranks are independent of the number of processes in the ring and the location of the process in the ring. These numbers are not necessarily consecutive; the individual processes do not know a priori how many processes are in the ring.

Figure 6-5 A ring network.

Figure 6-6 A ring network with ranks.

The ring is not static. Instead, processes enter and leave the ring dynamically. The logical ring is automatically (through the underlying hardware) patched to handle changes in the ring's configuration.

The departure of the king process from the ring requires an election to crown the new king. An *election* is an algorithm that determines the process that ought to be the king by passing messages containing the ranks of the active processes around the ring. For the sake of simplicity, we assume that no process leaves or enters the ring during an election.

Our naive election algorithm has each process assign its own number to the variable shared with its "left" neighbor and read from its "right" shared variable its right neighbor's rank. Processes forward, from right to left, any rank that is larger than anyone they have already seen. When a process receives its own rank back through its right variable, it knows that that value has passed all the way around the ring, making that process the legitimate chief—the lord of the ring. The new king process announces its election by circulating the negative of its rank around the ring. The program for election algorithm is as follows:

```
process candidate;
shared var left, right: integer (initial 0);      -- imported variables
const MyNum = 1729;

function election: boolean;      -- returns true if this process is elected king
var best, last: integer;
begin
    left   := MyNum;
```

```
        best := MyNum;      -- the best this process has seen so far
        while (right $\geq$ 0) and not (right = MyNum) do
            if (right > best) then
            begin               -- a new, higher rank
                last  := right;
                left  := last;
                best  := last
            end;
        election := (right = MyNum);
        if election
            then  left := −MyNum
            else  left := right;
    end; -- election
                ⋮
      -- the rest of process candidate
                ⋮
    end -- candidate
```

If there are n processes in the ring, this algorithm may make $O(n^2)$ assignments to the shared variable. Figure 6-7 shows an example of an arrangement of process ranks that can yield this worst case. In the figure, the ranks are sorted in decreasing clockwise order. The worst-case behavior happens when each process sends its value as far as possible to the left without sensing the higher rank on its right. That is, the lowest ranking process sends its message one step to the left. Then the second lowest process sends its message two steps left. This continues until the king process finally awakens and transmits its rank. This requires $1 + 2 + 3 + \cdots + n = n(n + 1)/2$ messages to be sent.*

This is not the most efficient possible solution. Hirschberg and Sinclair [Hirschberg 80] have shown an algorithm that solves the election problem in at worst $O(n \log n)$ writes. Their algorithm relies on writing values that contain a limit on how far around the ring they can travel. The processes send these messages around the ring in alternating directions until two messages from the same source meet.

Continuous display The strongly asynchronous nature of the Shared Variables model makes the system ideal for modeling Kieburtz and Silberschatz's continuous display problem [Kieburtz 79]. The *continuous display problem* hypothesizes a system of two processes, the generator and the display. The system tracks and displays. The *generator* continuously discovers the current location of some moving object, the *target*, and passes that location to the display process. The

* Our algorithm requires another n writes to circulate the announcement of the election.

Figure 6-7 The worst case of the election algorithm.

display process uses that value to show the "most current" location of the object on a display device. This idea of using the most recently generated value (and disregarding earlier values) is appropriate for many real systems, such as tracking systems. Synchronous communication would slow the system down to the speed of the slower process; separating the two processes by a producer-consumer buffer ensures that the stalest data is always the next read. However, shared memory obtains precisely the desired behavior. The generator repeatedly writes the current target location in the shared variable; the display just reads the shared variable's current value.

Perspective

Lynch and Fischer assert that some analysis and correctness problems are best attacked by modeling a set of distributed processes as if they shared a small amount of common storage. They argue that this model is primitive; that other more complicated communication mechanisms (such as messages and procedures) can all be expressed with shared variables.

Lynch and Fischer are concerned with formal programming semantics. Their work is heavily mathematical (principally the situation calculus over automata, expressed in predicate logic) and only marginally concerned with programming practice. The shared-variable concept is a low-level approach. It requires that every detail of an interaction be explicit. As such, it is good for modeling low-level activity and for deriving complexity results. The model is an "automata theory" of distributed computing. However, just as Turing machines are not a good basis for sequential programs, shared variables are not an appropriate foundation for practical coordinated computing.

The techniques that have been used to analyze the time complexity of shared-variable programs can also be used to analyze of the complexity of other aspects of programs. For example, Burns et al. [Burns 80b] present an analysis of the shared space requirements of various algorithms.

PROBLEMS

† **6-1** Extend the election algorithm to allow processes to be added or deleted during the election. Take particular care to handle the deletion of the king process. Assume that if a process leaves the ring, the ring is patched around its place.

6-2 How do the solutions to the previous problem change if all processors have (roughly similar) clocks and failures can be detected through time-outs?

6-3 Does the election program need variable last?

6-4 What is the best case of the naive election algorithm and when does it occur?

6-5 Program a solution to the dining philosophers problem using shared variables.

6-6 Modify the protocol of the message transmission program so that instead of sending an end-of-message symbol, messages are transmitted with their length.

6-7 Modify the protocol of the message transmission program so that neither processor can starve, unable to send a message.

6-8 In the message transmission program, why must processes read an okay_to_send before beginning message transmission? Why is shared variable pq_var initialized to okay_to_send?

6-9 The protocol program is relatively inefficient for the number of bits of information that are written for the size of message transmitted. Improve it.

6-10 Show how the card reader of Chapter 8 can be done using the Lynch-Fischer shared-variable model.

REFERENCES

[**Burns 80a**] Burns, J. E., personal communication, 1980.

[**Burns 80b**] Burns, J. E., P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson, "Data Requirements for Implementation of *N*-Process Mutual Exclusion Using a Single Shared Variable," *JACM*, vol. 29, no. 1 (January 1982), pp. 183–205. This paper shows how the shared-variable model can be used to analyze the space requirements of communication.

[**Cremers 79**] Cremers, A., and